Fault Localization for Firewall Policies

JeeHyun Hwang[1] Tao Xie[1] Fei Chen[2] Alex X. Liu[2]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206

[2] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824-1266

jhwang4@ncsu.edu xie@csc.ncsu.edu feichen@cse.msu.edu alexliu@cse.msu.edu

Abstract

Firewalls are the mainstay of enterprise security and the most widely adopted technology for protecting private networks. Ensuring the correctness of firewall policies through testing is important. In firewall policy testing, test inputs are packets and test outputs are decisions. Packets with unexpected (expected) evaluated decisions are classified as failed (passed) tests. Given failed tests together with passed tests, policy testers need to debug the policy to detect fault locations (such as faulty rules). Such a process is often time-consuming. To help reduce effort on detecting fault locations, we propose an approach to reduce the number of rules for inspection based on information collected during evaluating failed tests. Our approach ranks the reduced rules to decide which rules should be inspected first. We performed experiments on applying our approach. The empirical results show that our approach can reduce 56% of rules that are required for inspection in fault localization.

1 Introduction

Serving as the first line of defense against malicious attacks and unauthorized traffic, firewalls are crucial elements in securing the private networks of businesses, institutions, and home networks. A firewall is typically placed at the point-of-entry between a private network and the outside Internet so that all network traffic has to pass through the firewall. As a firewall highly depends on the quality of its firewall policy in terms of packet filtering, firewall errors are mostly caused by faults (i.e., misconfigurations) in rules of firewall policies. Wool [10] studied a large number of real-life firewalls and found that more than 90% of them have faults in their rules.

Various firewall policy testing tools [3, 5–7] have been proposed to expose security problems of policies. In firewall policy testing, test inputs and outputs are packets and their evaluated decisions (against a firewall policy under test), respectively. Firewall policy testers inspect each packet-decision pair to check whether the decision is expected. The testers classify packet-decision pairs into passed tests and failed tests. Packets that are evaluated to expected decisions are classified as passed tests. Packets that are evaluated to unexpected decisions (due to faulty rules in a policy) are classified as failed tests. Failed tests indicate that the policy under test includes faults, which cause failures.

Given failed tests together with passed tests, policy testers often debug a firewall policy to fix a fault. When a firewall fails to filter a packet properly due to a fault in its policy, the policy testers debug the policy — analyze the failure, locate the fault, and fix the fault. Among the three steps, locating the fault (i.e., fault localization) requires time-consuming and mostly manual inspection of the policy to detect its fault locations (i.e., root causes) such as rules that cause the failure. When a firewall policy may consist of a nontrivial number of rules, locating faulty rules by manual inspection is an error-prone, time-consuming, and tedious task. Therefore, developing an effective approach for fault localization can greatly help the policy testers reduce the cost of debugging.

In this paper, we focus on fault localization for a firewall policy including a single fault. We model two important fault types, Rule Decision Change (RDC) and Rule Field interval Change (RFC). An RDC fault indicates incorrect decision in a rule and an RFC fault indicates incorrect range values in a field (e.g., source/destination IP addresses). When a policy includes such faults, some packets will be evaluated to incorrect decisions. Our approach first analyzes a faulty policy and its failed tests. The approach considers rules covered by failed tests and determines whether the covered rules include a fault such as RDC and RFC. If the covered rules do not include a fault, our approach suggests other likely faulty rules for inspection. To reduce effort for detecting fault locations, our approach reduces the number of rules for inspection based on faulty behaviors caused by a fault. To help further reduce effort, our approach ranks the reduced rules to decide which rules should be inspected first based on structural coverage

1060-9857/09 $26.00 © 2009 IEEE
DOI 10.1109/SRDS.2009.38

during evaluation of failed tests.

The rest of this paper is organized as follows. Section 2 presents background information on firewall policies, policy model, and structural policy coverage, and packet generation. Section 3 describes a fault model for firewall policies. Section 4 describes our approach. Section 5 describes the evaluation where we apply the approach on various firewall policies. Section 6 discusses related work. Section 7 concludes the paper.

2 Background

This section presents the background information for our approach, including an introduction of firewall policies as well as our previous work [3] on a firewall policy model, structural coverage measurement, and packet generation.

2.1 Firewall Policies

A firewall policy is composed of a sequence of rules that specify under what conditions a packet is accepted and discarded while passing between a private network and the outside Internet. In other words, the policy describes a sequence of rules to decide whether packets are accepted (i.e., being legitimate) or discarded (i.e., being illegitimate). A rule is composed of a set of fields (generally including source/destination IP addresses, source/destination port numbers, and protocol type) and a decision. Each field represents a set of possible values (to match the corresponding value of a packet), which are either a single value or a finite interval of non-negative integers.

A packet consists of values, each of which corresponds to a field. A packet *matches* a rule if and only if each value of the packet satisfies the corresponding values in the rule. Upon finding a matching rule, the corresponding decision of that rule is derived. When evaluating a packet, a firewall policy follows the first-match semantic: the decision of the first matching rule is the decision of the packet.

Figure 1 includes an example of a firewall policy. The symbol "*" denotes the corresponding domain range (e.g., 0.0.0.0 - 255.255.255.255 for the IP address field). The example has three firewall rules r_1, r_2, and r_3. Rule r_1 accepts any packet whose destination IP address is $192.168.*.*$ (which indicates the range [192.168.0.0, 192.168.255.255]). Rule r_2 discards any packet whose source IP address is $1.2.3.*$ (which indicates the range [1.2.3.0, 1.2.3.255]) and port is the range $[1, 2^8-1]$ with the TCP protocol type. Rule r_3 is a tautology rule to discard all packets. Let k be a packet whose destination IP address is 192.168.0.0 and protocol type is UDP. When evaluating k, we find that k can match both r_1 and r_3. Among the two rules, as r_1 is the first-matching rule, k is evaluated to be accepted (based on the decision of r_1).

2.2 Firewall Policy Model

In this paper, we use a model of a firewall policy based on common generic features [3]. A firewall policy is composed of a sequence of rules, each of which has the form (called the generic representation) as follows.

$$\langle \textit{predicate} \rangle \rightarrow \langle \textit{decision} \rangle \qquad (1)$$

A $\langle \textit{predicate} \rangle$ in a rule defines a set of packets over a finite number of *fields* F_1, ..., F_n. The $\langle \textit{decision} \rangle$ of a rule can be *accept* or *discard* and returned as the evaluation result when the $\langle \textit{predicate} \rangle$ is evaluated to be true.

A packet k can be viewed as a tuple (fv_1, ..., fv_n) over a finite number of *fields* F_1, ..., F_n, where fv_i is a variable whose values are within a domain, denoted by $D(F_i)$. In a policy model, we convert values in *fields* to corresponding integer values to simplify a representation format (e.g., IP address 255.255.255.255 is $2^{32}-1$). Let S_i denote a subset of domain $D(F_i)$. The $\langle \textit{predicate} \rangle$ can be represented as

$$(F_1 \in S_1) \wedge ... \wedge (F_n \in S_n) \qquad (2)$$

We refer to each $F_i \in S_i$ as a $\langle \textit{clause} \rangle$, which can be either evaluated to true or false. Note that a firewall policy follows the first-match semantic. Given a sequence of rules, the following process is iterated until reaching the last rule: if a $\langle \textit{predicate} \rangle$ in a rule is evaluated to true, then the corresponding decision is returned; otherwise, the next rule (if exists) is evaluated.

2.3 Coverage Measurement

We have defined three types of policy structural coverage for each of the three major entities in a firewall policy: rules, predicates, and clauses [3].

- *Rule coverage.* A rule is covered by a packet if the rule is the first matching rule on the packet and the decision of the rule is derived.
- *Predicate coverage.* A predicate can be evaluated to true or false. A true (false) predicate is covered by a packet if the predicate is evaluated to true (false).
- *Clause coverage.* A clause can be evaluated to true or false. A true (false) clause is covered by a packet if the clause is evaluated to true (false).

To automate the measurement of policy coverage, we use a measurement tool [3] to evaluate packets against the policy under test.

2.4 Packet Generation Based on Local Constraint Solving

In our previous work [3], we developed automated packet generation techniques based on constraint solving.

Rule	Source IP	Source Port	Destination IP	Destination Port	Protocol	Decision
r_1	*	*	192.168.*.*	*	*	accept
r_2	1.2.3.*	*	*	$[1, 2^8 - 1]$	TCP	discard
r_3	*	*	*	*	*	*discard*

Figure 1. An example firewall policy

Among the choices of packet generation techniques, we observed that packets generated by solving individual rule constraints can achieve comparable coverage with that by solving combined constraints, but requires much lower analysis cost. Therefore, our approach uses packet generation by solving individual rule constraints (rather than that by solving combined constraints).

This technique statically analyzes rules to generate test packets. Given a policy, the packet generator first analyzes the entities in an individual rule and generates packets to evaluate the constraints (i.e., conditions) of the entities to be true or false. Second, the generator constructs constraints to evaluate each clause in a rule to be either false or true. Finally, the generator generates a packet based on the concrete values derived to satisfy each constraint by constraint solving.

3 Fault Model

This section describes potential faults in firewall policies. Assume that there is a single fault in a firewall policy. Based on fault models [3], we focus on two types of single faults:

- *Rule Decision Change (RDC)*. Faults in the RDC category indicate that Rule r_i's decision is incorrect. For example, the policy authors assign `discard` instead of `accept` (that is assumed correct) in r_i.
- *Rule Field interval Change (RFC)*. Faults in the RFC category indicate that a field interval F_i in Rule r_i is incorrect. For example, the policy authors assign "*" instead of a specific interval [0, 50] (that is assumed correct) in r_i's source IP address field.

These faults describe that a firewall rule contains a single fault in its structural entity (e.g., a decision or clause). Any fault in a structural entity is critical for compromising the correctness of policy behavior.

4 Fault Localization

For fault localization, we develop three techniques to reduce the number of rules for inspection and rank the reduced rules to decide which rules should be inspected first.

	# Rule Cov		Selected
	#Failed	#Passed	
R_1: $F_1 \in [0,10] \wedge F_2 \in [3,\ 5] \wedge F_3 \in [3,\ 5] \rightarrow$ *accept*	0	2	
R_2: $F_1 \in [5,\ 7] \wedge F_2 \in [0,\ 10] \wedge F_3 \in [3,\ 5] \rightarrow$ *discard*	0	2	
R_3: $F_1 \in [5,\ 7] \wedge F_2 \in [0,\ 10] \wedge F_3 \in [6,\ 7] \rightarrow$ *accept*	0	2	
R_4: $F_1 \in [2,10] \wedge F_2 \in [0,\ 10] \wedge F_3 \in [5,10] \rightarrow$ *accept*	2	0	●
R_5: $F_1 \in [0,10] \wedge F_2 \in [0,\ 10] \wedge F_3 \in [0,10] \rightarrow$ *discard*	0	2	

Figure 2. Example faulty policy with structural coverage and computed metrics about its test suite, and its rules' rank. In the policy, R_4's decision is incorrect and this faulty rule is in bold.

4.1 Covered-Rule-Fault Localization

This section describes a technique to locate a faulty rule when it is covered by failed tests. This technique is to detect a faulty rule in a policy by suggesting the policy testers to inspect the earliest-placed rule covered by failed tests. If many rules rs are covered by failed tests, the earliest-placed rule r_p refers to the rule that is located above other rules covered by failed tests and is given the highest priority among rs for inspection. This technique suggests r_p as a faulty rule for inspection; other rules in rs may be covered (by failed tests) due to a fault in r_p.

Let a policy p include a sequence of rules, r_1, r_2, ..., r_n. Rule r_i refers to the faulty rule in p. Consider that rules r_i and r_j are covered by failed tests k_i and k_j, respectively. If rule r_i located below r_j (i > j), no matter how we change the predicate or decision of r_i, the failed test k_j still matches or covers r_j and its decision is still incorrect.. Therefore, the faulty rule r_i should be located above r_j (i ≤ j) and should be inspected to fix. Among r_i and r_j, r_i is the earliest-placed rule.

Considering an example faulty firewall policy in Figure 2, the numbers in Columns 2 and 3 show the times of each rule covered by failed and passed tests, respectively. In this example, 10 tests are used: 8 tests are passed and 2 tests are failed. We inject an RDC fault to R_4 by changing its decision from "discard" to "accept". According to our technique, because R_4 is the earliest-placed rule covered by failed tests, we suggest that R_4 is likely to contain a fault for inspection.

	# Rule Cov		# Clause Cov by failed tets			Selected	Rank
	#Failed	#Passed	#True	#False	Susp		
R_1: $F_1\in[0,10]\wedge F_2\in[3, 5]\wedge F_3\in[3, 3]\rightarrow$ *accept*	0	2	6	3	0.66	●	2
R_2: $F_1\in[5, 7]\wedge F_2\in[0, 10]\wedge F_3\in[3, 5]\rightarrow$ *discard*	0	2					
R_3: $F_1\in[5, 7]\wedge F_2\in[0, 10]\wedge F_3\in[6, 7]\rightarrow$ *accept*	0	2	4	5	0.44	●	3
R_4: $F_1\in[2,10]\wedge F_2\in[0, 10]\wedge F_3\in[5,10]\rightarrow$ *discard*	2	1				●	1
R_5: $F_1\in[0,10]\wedge F_2\in[0, 10]\wedge F_3\in[0,10]\rightarrow$ *discard*	1	2					

Figure 3. Example faulty policy with structural coverage and computed metrics about its test suite, and its rules' rank. In the policy, range values in field F_3 (of rule R_1) are incorrect and this faulty rule is in bold.

4.2 Rule Reduction

If the earliest-placed rule covered by failed tests does not include a fault, we propose two techniques to effectively reduce the number of rules for inspection.

The first technique is based on that a faulty rule is located at least above or at the earliest-placed rule by failed tests. The technique suggests rules (that are located at least above or at the earliest-placed rule covered by failed tests) as likely faulty rules for inspection. Let a policy p includes a sequence of rules, r_1, r_2, ..., r_n. Consider that r_i is the earliest-placed rule covered by failed tests during evaluation of failed tests. If rule r_j (j > i) is the faulty rule, r_i is given higher priority than r_j during evaluation. In such a case, no matter how we change the predicate or decision of r_j, packet k_i that covers r_i still has an incorrect decision. Therefore, the faulty rule r_j should be located at the same or above r_i (j $\leq$ i) to cause faulty policy behavior observed during evaluation of r_i.

The second technique is based on that a faulty rule's decision dec_j must be different from the decision dec_i of rules covered by failed tests. Among rules selected by the first technique, the technique further selects rules (whose decisions are different from the decision of rules covered by failed tests) as likely faulty rules for inspection. Suppose that a failed packet covers a rule r_i with decision dec_i, and a rule r_j (j < i) is the faulty rule (including a faulty clause) with decision dec_j. After we correct the faulty clause of r_j, the failed packet covers r_j. As dec_i is not the expected decision for the failed packet, dec_i and dec_j should be different; otherwise, the correction of the faulty clause cannot lead to the expected decision for the packet.

4.3 Rule Ranking

This section describes a technique to rank rules based on their likelihood of being faulty by considering clause coverage. The rationale behind the metrics for the rule ranking technique is based on that clause coverage of a faulty rule by failed packets often demonstrates some special characteristic; among rules selected by the previous technique, the number of clauses that are evaluated to false in a faulty rule is at least equal to or smaller than that in other rules. Therefore, the technique monitors clause coverage and ranks rules for inspection by the order of small ratios of false clauses over all clauses evaluated by all failed packets.

Assume that rule r_j includes a single RFC fault and a failed test packet k covers rule r_i rather than r_j (j < i). When a failed test evaluates r_j, a faulty clause c in r_j is evaluated to false by k; however, the remaining clauses in r_j are evaluated to true. In other words, only a *single* clause (which is the fault location) in r_j is evaluated to false by k. When k evaluates other rules that are located above r_i, k evaluates *at least* one clause to false not to match these rules. Therefore, for r_j, the number of false clauses evaluated by k could be relatively smaller than the number of false clauses for other faulty likely rules.

We develop an equation to measure the suspiciousness of rules, called *suspicions* based on clause coverage. For each rule, the *suspicion* is computed based on the number of false clauses during evaluation of failed tests divided by the number of true clauses covered by failed tests. The higher *suspicion* on r, a higher likelihood of r to be faulty. Our technique ranks rules by suspicions in a decreasing order. For each rule r, let *FF(r)* (*FT(r)*) be the number of false (true) clauses during evaluation of failed tests. For each rule r, we measure *suspicion (r)* with the following equation:

$$suspicion(r) = 1 - \frac{FF(r)}{FF(r) + FT(r)} \qquad (3)$$

Figure 3 shows an example faulty policy and its rule reduction. Columns 2-3 show the times of each rule covered by failed and passed tests, respectively. Columns 4-8 show the times of true and false clauses covered by failed tests, respectively, and suspicion based on clause coverage (Column "Susp"), selected rules (for inspection), and each rule's rank to decide which rules should be inspected first. In Figure 3, R_1 is a faulty rule because we inject an RFC fault by changing its third clause values from "[3,5]" to "[3,3]".

In this example, 12 tests are generated: 9 tests are passed tests and 3 tests are failed tests. In this example, R_4 and R_5 are covered by failed tests. We first inspect the the earliest-

Table 1. Real-life policies and faulty policies with at least one failed test.

Policy	# Rules	# Tests	# RDC	# RFC
1 (fw.LAN)	28	140	14	50
2 (fw.MAIL1SLU)	18	90	7	42
3 (fw.MAILHOST)	26	130	16	60
4 (fw.MAILHOST2)	26	130	11	61
5 (fw.MAILHOST3)	27	135	12	63
6 (fw.MAILHOST4)	28	140	22	77
7 (fw.NEWS)	14	70	9	28
8 (fw.NS3)	17	85	9	29
9 (fw.RCPROTO1)	23	115	10	42
10 (fw.SSH)	16	80	6	26
11 (fw.WAN)	24	120	15	42
12 (fw.std4)	42	210	31	58
13 (fw.std-format5)	87	435	86	304
14 (fw.std0)	661	3305	637	3113
Average	74.07	370.36	63.21	285.36

placed rule R_4 covered by failed tests and discover that R_4 does not contain a fault.

If we cannot locate faults by inspecting R_4, we apply the rule reduction technique in Section 4.2. We consider three rules R_1, R_2, R_3 located above R_4 for inspection; these rules are given higher priority than R_4 during evaluation, and a fault in one of these rules can affect R_4 to introduce incorrect policy behavior. Among the three rules, we select R_1 and R_3 as likely faulty rules because their decisions (“accept”) are different from R_4’s decision (“discard”).

Considering R_1 and R_3 in Figure 3, *suspicion* (R_1) is 0.66 (1 - 3/9 = 0.66) and *suspicion* (R_3) is 0.44 (1 - 5/9 = 0.44). Therefore, R_1 is given a higher rank for inspection than R_3.

5 Evaluation

We conducted our experiments on a laptop PC running Windows XP SP2 with 1G memory and dual 1.86GHz Intel Pentium processor. Experimental subjects include faulty policies (with their failed and passed tests) synthesized from real-life firewall policies. For each faulty policy and its failed/passed tests, our tool first analyzes rules covered by failed tests and checks whether the earliest-placed rule include a fault. If the earliest-placed rule covered by failed tests does not include a fault, the tool further reduces the number of rules for inspection based on our proposed rule reduction techniques. To further reduce effort, our tool ranks the reduced rules to decide which rules should be inspected first.

5.1 Subjects

Our experimental subjects include 885 RDC and 3995 RFC faulty policies synthesized from 14 real-life firewall policies collected from a variety of sources. Our tool first analyzes a given real-life policy and synthesizes its faulty policies by seeding a fault on each clause or decision. We call a faulty policy a mutant and each mutant includes one seeded fault. The tool generates test packets using the packet generation technique based on local constraint solving [3] (for each faulty policy). The tool compares two decisions for a packet: the decision evaluated against a faulty policy and the decision evaluated against its original policy. If two decisions are inconsistent (consistent), the tool classifies such a test packet as a failed (passed) test. Among our generated mutants, we select *only* faulty policies with at least one failed test.

Table 1 shows the basic statistics and the number of synthesized faulty policies for each real-life policy. Columns 1-2 show the policy names and the numbers of rules for each real-life policy, respectively. Column 3 shows the number of generated tests. Columns 4-5 show the number of generated RDC and RFC faulty policies, respectively, with at least one failed test.

5.2 Objectives and Measures

In the experiments, we intend to answer the following research questions:

1. Can our covered-rule-fault localization detect a faulty rule by inspecting the earliest-placed rule covered by failed tests?
2. Can our rule reduction technique effectively reduce the number of rules for inspection?
3. Can our rule ranking technique further improve the results of the rule reduction technique in terms of rule reduction percentage for inspection?

Intuitively, without effective tool support, testers inspect all rules. Averagely, if rules are randomly or arbitrarily selected for inspection, half of rules could be inspected until a faulty rule is detected. Let the number of rules be r. The number of inspected rules on average is $\overline{r} = \frac{r}{2}$. If rules are ordered by our rule ranking technique for inspection, a faulty rule’s rank n refers to the number of inspected rules until the faulty rule is detected. For each mutant and its test packets, we measure the following metrics:

- *Rule reduction percentage*. Let the total number of rules be r and the number of the reduced rules be r'. We define the rule reduction percentage, $\%Reduction$, as follows:

$$\%Reduction = (1 - \frac{\overline{r'}}{\overline{r}}) \times 100 = (1 - \frac{\frac{r'}{2}}{\frac{r}{2}}) \times 100$$

- *Ranking-based rule reduction percentage*. Let the total number of rules be r and the rank of a faulty rule be n.

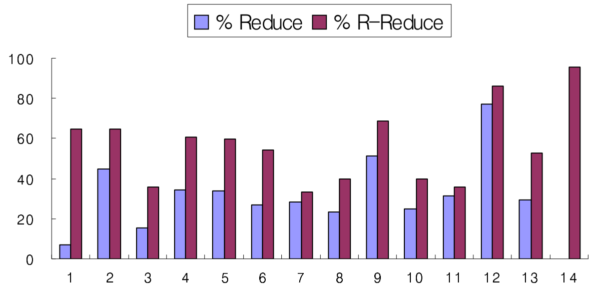

Figure 4. Rule reduction and ranking-based rule reduction percentages for each synthesized policy group

We define the ranking-based rule reduction percentage, $\%R\text{-}Reduction$, as follows:

$$\%R{-}Reduction = (1 - \frac{n}{r}) \times 100 = (1 - \frac{n}{\frac{r}{2}}) \times 100$$

- *Passed-test count.* The passed-test count is the average number of passed packet-decision pairs for policies synthesized based on a given policy.
- *Failed-test count.* The failed-packet count is the average number of of failed packet-decision pairs for policies synthesized based on a given policy.

5.3 Results

We observed that 100% of RDC faulty rules are covered by at least one failed test. As failed tests cover an RDC faulty rule, the rule coverage information (i.e., the covered rules) is sufficient to locate such faults. To detect such faults, policy testers are required to inspect the earliest-placed rule among covered rules during evaluation of failed tests. the earliest-placed rule is suggested by our covered-rule-fault localization technique. We observed that 69% of RFC faulty rules are covered by at least one failed test. In such a case, among covered rules by failed tests, policy testers are required to inspect the earliest-placed rule suggested by our covered-rule-fault localization technique.

We next present results about faulty rules not being covered by failed tests. 31% of RFC faulty rules are not covered by only failed tests. Table 2 shows the basic statistics of such faulty policies used in our experiments and their rule reduction percentage on average by our proposed techniques: rule reduction and rule ranking techniques. Columns 1-2 show the policy names and the number of generated RFC faulty policies for each real-life policy. Columns 3-4 show the average number of passed and failed tests, respectively, generated based on each real-life policy. Columns 5-6 show the average percentage of rule reduction for each real-life policy by our rule reduction and ranking techniques, respectively. Figure 4 illustrates rule reduction percentage comparison of these two techniques for the faulty policies shown in Table 2. The x axis shows the indexes of the policy names for faulty policies and the y axis shows the average rule reduction percentage.

In Table 2, we observed that there is a correlation between the structure of a firewall policy and its rule reduction percentage. One example is the `fw.std0` policy. The policy consists of 661 rules; 660 rules are specified to accept some packets and 1 remaining rule is a tautology rule to discard all other packets. As types of decisions are too imbalanced in this policy, the rule reduction technique cannot reduce rules based on selecting rules including different decisions from that of its earliest-placed rule. For the policy, we observed that 0% of rules are reduced for inspection. However, the rule ranking technique is not affected by such imbalanced decisions in the policy. We observed that this policy achieves the highest reduction percentage, 95.72% considering clause coverage.

We also observed that 30.63% of rules on average are reduced for inspection (Column "% Reduction"). Our technique is effective to reduce 30.63% of rules required for inspection. Such a reduced number of rules help policy testers to reduce their inspection efforts. Based on our ranking technique, we observed that 56.53% of rules on average are reduced for inspection (Column "% R-Reduction"). The ranking further achieves around 26% reduction for rule inspection in comparison with the results of the rule reduction. Overall, our ranking further improves the rule reduction technique.

6 Related Work

Various techniques have been proposed on fault localization of software programs in software engineering and programming language communities [1, 2, 4]. These techniques aim to help a developer find faulty code locations in a program quickly. These techniques are based on dynamic program slicing [1], the nearest neighbor technique [9], and statistical techniques [4]. These preceding techniques typically analyze the likely faulty locations based on dynamic information collected from running the faulty program. Particularly, statistical techniques use data collected from failing and successful runs of a program to find likely faulty locations. Firewall polices and general programs are fundamentally different in terms of structures, semantics, and functionalities, etc. Therefore, fault localization techniques of general programs are not suitable for addressing the fault localization problem of firewall policies.

Prior work that is closest to ours is Marmorstein et al.'s work [8]. They proposed a technique to find failing packets that violate the requirement specification of a firewall policy, and presented another technique to further use the failing packets and the history map of firewall rules to easily identify two or three faulty rules in a firewall policy. Our proposed techniques have three main advantages over their

Table 2. Faulty policies (including an RFC faulty rule un-covered by its failed tests) used in the experiment.

Policy	# RFC	# Avg-Passed	# Avg-Failed	% Reduction	% R-Reduction
1 (fw.LAN)	18	136.11	3.89	7.14	64.68
2 (fw.MAIL1SLU)	12	85.25	4.75	44.91	64.81
3 (fw.MAILHOST)	17	128.06	1.94	15.38	35.97
4 (fw.MAILHOST2)	17	126.18	3.82	34.39	60.86
5 (fw.MAILHOST3)	17	131.24	3.76	33.99	59.48
6 (fw.MAILHOST4)	30	138.07	1.93	26.90	54.17
7 (fw.NEWS)	9	68.44	1.56	28.57	33.33
8 (fw.NS3)	5	84.00	1.00	23.53	40.00
9 (fw.RCPROTO1)	8	113.50	1.50	51.09	68.48
10 (fw.SSH)	3	78.33	1.67	25.00	39.58
11 (fw.WAN)	20	117.80	2.20	31.46	35.83
12 (fw.std4)	16	209.00	1.00	77.08	85.86
13 (fw.std-format5)	176	432.68	2.32	29.43	52.59
14 (fw.std0)	738	3303.95	1.05	0.00	95.72
Average	77.57	368.04	2.31	30.63	56.53

work. First, their work misses all the potential faulty rules that we handled in the RFC fault category because they consider only the rules that match a failed packet whereas we examine the rules above the rule that failed packets match. Second, we reduced the number of possible faulty rules, while their work can only find out all the potential faulty rules that cover the failed packets. Third, we ranked the potential faulty rules to facilitate searching of the faulty rule, while their work did not rank rules.

7 Conclusions and Future Work

We have developed a fault localization approach for a firewall policy where a Rule Decision Change (RDC) or Rule Field interval Change (RFC) fault exists. Our empirical results showed that 100% of RDC faulty rules and 69% of RFC faulty rules can be detected by inspecting covered rules. However, 31% of RFC faulty rules are not covered by only failed test. We further applied our reduction and ranking techniques on such faulty policies. Our empirical results showed 30.63% of rules on average are reduced for inspection based on our rule reduction technique and 56.53% of rules on average are reduced for inspection based on our rule ranking.

Our approach has been shown to be practically effective to locate a single fault in a firewall policy. However, faults in a policy may consist of one or several simple faults. We plan to extend our approach to handle a policy containing multiple faults in future work.

Acknowledgment

This work is supported in part by NSF grant CNS-0716579, NSF grant CNS-0716407, and MSU IRGP Grant.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[2] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. of the 27th International Conference on Software Engineering*, pages 342–351, 2005.

[3] J. Hwang, T. Xie, F. Chen, and A. X. Liu. Systematic structural testing of firewall policies. In *Proc. of the 27th IEEE International Symposium on Reliable Distributed Systems*, pages 105–114, 2008.

[4] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.

[5] J. Jürjens and G. Wimmel. Specification-based testing of firewalls. In *Proc. of the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 308–316, 2001.

[6] A. X. Liu, M. G. Gouda, H. H. Ma, and A. H. Ngu. Non-intrusive testing of firewalls. In *Proc. of the 1st International Computer Engineering Conference*, pages 196–201, 2004.

[7] M. R. Lyu and L. K. Y. Lau. Firewall security: Policies, testing and performance evaluation. In *Proc. of the 24th International Conference on Computer Systems and Applications*, pages 116–121, 2000.

[8] R. Marmorstein and P. Kearns. Assisted firewall policy repair using examples and history. In *Proc. of the 21st Conference on Large Installation System Administration Conference*, pages 1–11, 2007.

[9] M. Renieris and S. P. Reiss. Fault localizationwith nearest neighbor queries. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, 2003.

[10] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.